Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines

Philip Levis[†], David Gay[‡], and David Culler[†]
{pal,culler}@cs.berkeley.edu, dgay@intel-research.net

[†]EECS Department
University of California, Berkeley
Berkeley, CA 94720

[‡]Intel Research Berkeley
2150 Shattuck Avenue
Berkeley, CA 94703

ABSTRACT

We propose application specific virtual machines as a method to safely and efficiently program sensor networks. Although sensor networks encompass a wide range of application domains, any given network supports a single one. A VM tailored to a particular deployment can provide retasking flexibility within its application class while keeping programs efficient. We present Maté, an architecture for customizing VMs over a wide range of sensor network applications. Customizing the instruction set and triggering events allows for language flexibility, provides very high code density, and enables a wide range of applications.

We evaluate Maté by comparing custom built VMs to two existing proposals for user-level sensor network programming, abstract regions and tree-based aggregation (TinyDB). We show that a VM implemented in our architecture can provide equivalent functionality to the current implementations of these proposals while improving efficiency. Additionally, by decomposing application domains into a set of reusable, fine-grained software components, implementing new user-level programming abstractions is greatly simplified.

1. INTRODUCTION

Operating systems research has a long history of designing flexible abstractions. Sensor networks provide a new set of challenges and opportunities in this area: each deployment features specific hardware features and processing capabilities and, once deployed, networks must be reprogrammable in response to observed data or evolving needs. As each deployment targets at a specific application domain (e.g., habitat monitoring or tracking), the system environment can be tailored to the domain's needs.

In-situ reprogramming should be efficient and safe. Energy is the limiting resource in a sensor network: installing new programs and executing them must be energy efficient. As sensor networks are typically dense fields of small nodes embedded in the environment, recovering from a program-induced crash or failure – through rebooting, for example – is difficult or impossible. A programming system must be able to safely recover from buggy or badly conceived programs.

The need for higher-level sensor network programming has led to several proposed programming models. For example, abstract regions showed how data-parallel operators can concisely represent applications that require inter-node data aggregation [25], and TinyDB demonstrated the effectiveness of declarative SQL-like queries for data collection [18]. While these proposals are steps in the right direction, they address a limited set of application domains, and do not address all of the above requirements. TinyDB (SQL) is not a very general programming model and its query interpretation (i.e., program execution) is inefficient. Abstract regions compiles programs into an full TinyOS image and installs the entire executable (tens of kilobytes); program installation is inefficient and there is no safety.

Application specific virtual machines provide a way of capturing application building blocks while allowing flexible and dynamic composition at the operating system level. In this paper, we propose using application specific virtual machines as an intermediate layer between an application domain and the mote operating system (TinyOS). To further this goal, we have built Maté, a general architecture for application specific VMs. A user tailors a Maté VM to a specific application domain by selecting a programming language (in which end-users will program) and a set of domain-specific extensions (e.g., support for planar feature detection). The Maté framework compiles programs to the generated VM's instruction set (*bytecodes*), and the VM runtime automatically propagates code through the network. Installation-time code analysis allows a VM to provide race-free and deadlock-free execution parallelism.

The separation between the three layers – user pro-

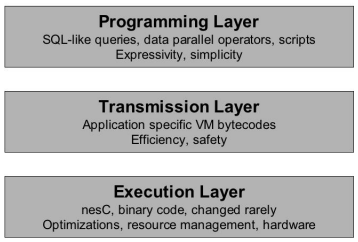

Figure 1: A layered decomposition of in-situ reprogramming.

gram, in-network representation, and the execution engine – is shown schematically in Figure 1. Using bytecodes as a transmission layer provides safety; tuning the level of abstraction (by selecting an appropriate language and set of extensions) to a particular application's requirements leads to efficient execution, as overhead can be minimized; compiling to application-specific bytecodes makes programs small (tens or a few hundred bytes), keeping propagation efficient.

Maté is general enough to support existing proposals for sensor network programming models. We have built a VM with extensions for region-based operations, in which typical programs are on the order of seventy bytes (Section 4.4), a 99.5% reduction in size over the originally proposed regions implementation. We have built a VM for performing similar sensor network queries to TinyDB (Section 4.5), with 11-30% energy savings. We have implemented two languages on top of Maté, TinyScript, a simple BASIC-like language, and motlle, a Scheme-like language, and believe the engine general enough to support other programming languages, such as TinySQL (our querying VM's language is motlle). These increases in efficiency come from separating the three layers in Figure 1. Separating the transmission layer from the execution layer reduces the cost of installing a regions program; separating the transmission layer from the programming layer reduces the cost of query execution.

In earlier work, we proposed a particular virtual machine as a mechanism to reprogram sensor networks [14], but it imposed draconian limitations that ultimately made it unusable. This paper extends the work to a general framework that supports multiple languages, programming models, and can be applied to a wide range of applications.

Section 3 covers the design of the Maté architecture. Section 4 presents and evaluates our VMs in comparison to regions (on a pursuer-evader application) and to TinyDB (on various data collection queries). We conclude this paper with a discussion of the implications of these results (Section 5), a survey of related work (Section 6) and areas for future work (Section 7).

```
module Main {
  uses interface StdControl as SubControl;
  ...
}
module MateEngineM {
  provides interface StdControl as Control;
  uses interface SendMsg as SendError;
  ...
}
configuration GenericComm {
  provides interface SendMsg[uint8_t id];
  ...
}
configuration MateTopLevel {
  components Main, MateEngineM as Engine;
  components GenericComm as Comm;
  Main.SubControl -> Engine.Control;
  Engine.SendError -> Comm.SendMsg[AM_ERROR];
  ...
}
```

Figure 2: nesC Wiring Examples. ***MateTopLevel wires together Main, MateEngineM and GenericComm. GenericComm provides SendMsg as a parameterized interface, where the parameter is the active message ID. Wiring Engine.SendError to GenericComm.SendMsg requires specifying the message type.***

2. BACKGROUND

TinyOS is a popular sensor network operating system than runs on a range of limited resource devices ("motes"). As motes need to be able to operate unattended on small batteries for the better part of a year, minimizing energy costs greatly influences their design. Correspondingly, hardware resources are very limited. Typical TinyOS motes have a 4-8MHz microcontroller, 4kB of data RAM, 60-128kB or program flash memory, and a radio with application-level data transmission rates of 1-2kB/s.

Energy limitations force sensor networks operate at very low utilization. Therefore, although a mote has very limited resources, for many application domains this is not a significant limitation. For example, in the Great Duck Island deployments [20], motes woke up every eight minutes, warmed up sensors for a second, and transmitted a single data packet with readings. During this second, the CPU was essentially idle. The one exception to this trend of low utilization is RAM. However, RAM limitations seem to be more a result of the market for current commercial microcontrollers than fundamental technical issues. Although much larger amounts (i.e., megabytes) would have significant energy costs, it seems likely future motes will feature significantly more data memory.

The nesC language [8], used to implement TinyOS and its applications, provides two basic abstractions: a component-based programming model, and a low-overhead, event-driven concurrency model. *Components* are the units of program composition. A component has a set of *interfaces* it requires, and a set of interfaces it provides. A programmer builds an application by wiring interface providers to requirers (see Figure 2). nesC sup-

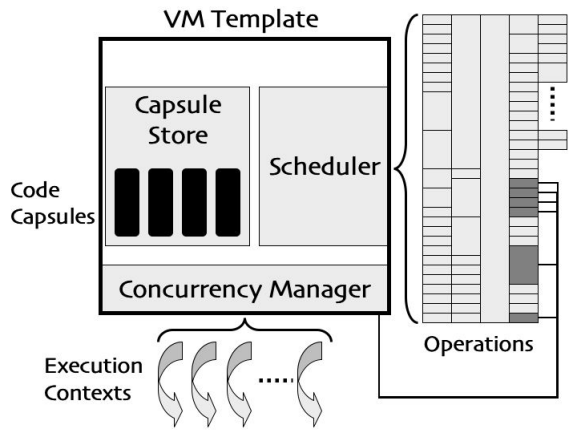

Figure 3: The Maté architecture.

ports two kinds of components, modules and configurations. A module represents actual program logic (i.e., an implementation); a configuration is a *wiring* of subcomponents. Configurations allow subsystem encapsulation. For example, the TinyOS component GenericComm encapsulates the entire networking stack (20 or so components) and provides just three interfaces, for power management, packet reception, and packet transmission. In addition to basic interfaces, nesC has parameterized interfaces. Essentially, a component can provide many copies of an interface instead of a single one, and these copies are distinguished by a parameter value instead of by name. These parameterized interfaces support runtime dispatch between a set of components.

TinyOS's event-driven concurrency model does not allow blocking. Hence, calls to long-lasting operations, such as sending a packet, are typically split-phase: the call to begin the operation returns immediately, and the called component signals an event to the caller when the operation completes.

3. DESIGN

Maté's principal goal is to define a flexible architecture for building application-specific sensor network scripting environments. An environment has two parts: a programming language for sensor network users, and a virtual machine bytecode interpreter on the motes to execute user programs. Unlike virtual machines such as the JVM [17], the goal is not to provide a fixed abstraction boundary, instead the goal is to allow sensor network developers to define the boundary at a level suitable to a particular application domain.

Figure 3 shows a functional decomposition of the Maté VM architecture. Maté VMs have three major abstractions: contexts, operations, and capsules. Contexts are the units of concurrent execution, operations are the units of execution functionality, and capsules are the units of code propagation. A VM's components fall into two classes: the components every VM includes (the basic template), and the components that define the particular Maté instance, tailored to a particular language and application domain.

The basic VM template includes the scheduler, concurrency manager, and capsule store. The scheduler executes runnable contexts in a FIFO round-robin fashion. The concurrency manager submits contexts to the scheduler based on whether they are ready to run and can safely access the shared resources they require. Unlike TinyOS, the scheduler and concurrency manager support blocking operations. The capsule store manages capsule storage and loading; it propagates capsules through the network and notifies higher level components when it receives new code.

The basic template does not include any data storage components. These are provided by the selected language. Maté defines a set of standard types (currently integer and sensor reading) and a stack used to pass values to/from functions (see below). It is possible to add additional types to Maté, though it is currently hard to do in a language-independent fashion. We expect to add improved support for type extension.

A specific VM instance wires a set of contexts and operations to the basic template. The set of contexts defines the events that trigger VM execution. The set of operations defines the VM instruction set. Every VM bytecode maps to an operation component, which implements the corresponding operation through the MateBytecode interface, shown in Figure 4. Choosing a language for the VM selects a set of operations, known as primitives, which provide the basic features needed by the language, such as accessing a variable or arithmetic.

Selecting a set of appropriate contexts and functions tailors a VM to an application domain. Functions are operations that take their arguments from, and return their result to, the Maté stack and which are generally language independent.[1] Examples include functions to control timers or obtain sensor readings. Functions are invocable via their bytecode (if they have one) or by other, language-specific mechanisms. A context is a language-independent component which triggers the execution of a *handler* in response to some event (e.g., packet reception). A handler is a sequence of bytecodes stored in a capsule. The actual mapping from handlers to capsule code sequences is language-specific.

The rest of this section presents the three core components (scheduler, concurrency manager, and capsule store) and their interaction with the application-specific contexts and operations in greater depth. In particular, we show how the concurrency manager provides race-

[1] Languages may also include their own functions.

```
interface MateBytecode {
  command result_t execute(uint8_t opcode,
                           MateContext* context);
}
module MateEngineM {
  uses interface MateBytecode as Code[uint8_t code];
  ....
  result_t execute(MateContext* context) {
    ... fetch the next opcode ...
    // and execute it via a parameterized interface
    call Bytecode.execute[op](op, context);
  }
}
configuration MateTopLevel {
  components MateEngineM as VM, OPgetvar4;
  components OPadd, OPsend, OPhalt, OPsettimer;

  VM.Code[OP_ADD]      -> OPadd;
  VM.Code[OP_SEND]     -> OPsend;
  VM.Code[OP_HALT]     -> OPhalt;
  VM.Code[OP_SETTIMER] -> OPsettimer;
  VM.Code[OP_GET]      -> OPgetvar4;
  VM.Code[OP_GET+1]    -> OPgetvar4;
  ...
  VM.Code[OP_GET+7]    -> OPgetvar4;
}
```

Figure 4: Maté scheduler and interfaces

free, deadlock-free execution of contexts that access shared resources. We conclude with an example of building a simple VM for region-based programming.

3.1 Scheduler: Execution Model

The core of the Maté architecture is a simple FIFO scheduler. This scheduler maintains a queue of runnable contexts, and interleaves their execution at a very fine granularity (every few operations). The scheduler executes a context by fetching its next bytecode from the capsule store, and dispatches to the corresponding operation component; a typical VM may have a hundred or so such components. Figure 4 contains nesC code snippets showing this structure, which allows the core of the VM to be independent of the particular instruction set it implements.

A component implementing a context C starts C in response to some event by submitting C to the concurrency manager. Operations that wish to halt or pause C also make their requests via the concurrency manager (see below). All requests to add or remove a context from the run queue come from the concurrency manager.

The ability to pause a context allows functions that encapsulate a split-phase TinyOS call to present a synchronous interface to the Maté programmer. When such a function executes, it pauses the current context via the concurrency manager. When the TinyOS completion event fires, the function's component resumes the context via the concurrency manager, which submits it to the scheduler when it decides it can run race free.

3.2 Concurrency Manager: Parallelism

Traditionally, the default behavior for concurrent programming environments (e.g., threads, device drivers) is to allow race conditions, but provide synchronization primitives for users to protect shared variables. This places the onus on the programmer to protect shared resources. In return, the skilled systems programmer can fine tune the use of the primitives for maximum CPU performance.

Maté takes the opposite approach, because embedded systems are event driven, more difficult to debug, and demand greater robustness. When a VM installs a new capsule, it runs a conservative program analysis to determine the set of shared resources used by the capsule's handlers and, hence, what resources their corresponding contexts will need. The Maté concurrency model is based on statically named resources, such as shared variables. Operations specify the shared resources that they use, but the analysis that determines a handlers complete resource usage is language-specific. The most conservative form of analysis assumes all handlers share resources – the motlle language takes this approach. However, this precludes possible parallelism. Section 4.1 describes TinyScript's resource model and analysis.

In the simplest instantiation of the model, a context acquires all of its resources when it begins and releases all of them when it ends. When the concurrency manager receives a request to run a context, it checks if the resources the context will require are available. If so, it submits the context to the scheduler. If not, it places the context on a wait queue. When a context releases resources, the concurrency manager checks the wait queue and submits any now-runnable contexts to the scheduler. As sensor network applications typically have low utilization, starvation is not an issue.

Handlers can use *scheduling points* to improve parallelism. Certain VM operations, such as the yield function or functions that pause contexts, are scheduling points. A scheduling point has a set of resources R_s it releases and a set of resources A_s it acquires; A_s must be a subset of R_s. By default, $R_s = \emptyset, A_s = \emptyset$. A handler can temporarily relinquish a resource for the duration of a scheduling point by adding it to both R_s and A_s. It can permanently relinquish a resource by adding it to R_s only. Currently, neither TinyScript nor motlle support using scheduling points, but we have written assembly programs that use this functionality.

The inequality $A_s \subseteq R_s$ is sufficient for building a deadlock- and data-race- free scheduler, assuming that a user does not temporarily relinquish a resource at a scheduling point when it needs atomic access across that scheduling point. Race-free behavior is very simple: a context cannot access resources it does not hold, and two handlers that may access a resource cannot run concurrently. We include the proof of deadlock-freeness in the Appendix.

Dynamic code updates complicate race-free execution. Terminating a context because new new code has arrived could leave data in an inconsistent state. However. wait-

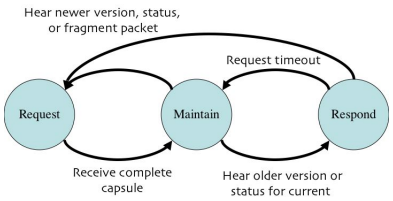

Figure 5: State diagram for Maté capsule propagation.

ing for it to complete may not be an option if the old version has an infinite loop. Therefore, when a new capsule arrives, the VM reboots, clearing out existing state. Applications that require persistent state can include functions that atomically store and load this state.

3.3 Capsule Store: Propagation

The initial Maté VM forwarded programs with an imperative `forw` instruction, which would broadcast code fragments. A single copy of a self-forwarding program would autonomously reprogram the network. However, this imperative forwarding had three major limitations. First, it was inefficient. Every node continued to transmit code even when the network was reprogrammed. Second, it could easily saturate the network. Without some sort of feedback or density estimation, a dense cluster of nodes would consume a lot of bandwidth endlessly broadcasting many copies of the same code. Finally, it tied propagation to execution. Handlers that ran rarely could not quickly forward themselves, and had to rely on others to do so.

The current Maté architecture solves these problems using the Trickle algorithm [16]. Trickle uses broadcast-based suppressions to quickly propagate new data but minimize overhead when nodes share data. Just as with explicit forwarding, once a user installs a single copy of a program in the network, Maté installs it on every mote. Pushing propagation into the VM runtime as a basic service means that users are not responsible for fine-tuning its performance, although a particular VM could include functions to manipulate propagation policies if an application required it.

Code propagation uses an epidemic-like approach: a node that has newer code will broadcast it to local neighbors. The Trickle algorithm is used to efficiently broadcast three entities: version packets, which contain the 32-bit version numbers of all installed capsules, capsule status packets which describes fragments of a capsule that a mote needs (essentially, a bitmask), and capsule fragments that are short segments of a capsule. Figure 5 contains the state diagram used by motes for code propagation. A mote can be in one of three states: maintain (exchanging version packets), request (sending capsule status packets), or respond (sending fragments). Nodes start in the maintain state. They enter the request state if they hear something that indicates someone has a newer capsule, whether it be a version, capsule status, or fragment packet. A requesting node returns to the maintain state once it receives the entire capsule. A node enters the respond state if it is in the maintain state and hears that someone has an older capsule (through a version packet), or needs part of its current capsule (through a capsule status packet). These state transitions mean that nodes prefer requesting over responding; a node will defer forwarding capsules until it thinks it is completely up to date.

Trickle's suppression operates on each type of packet (version, capsule status, and capsule fragment) individually. That is, a capsule fragment transmission will suppress all other fragment transmissions, but will not suppress version packets. This allows meta-data exchanges during propagation: sending a fragment will not cause someone to suppress a message saying what fragments it needs. Trickling fragments means that code propagates in a slow and controlled fashion, instead of as quickly as possible. This is unlikely to significantly disrupt any existing traffic, and prevents network overload. We show in Section 4.3 that because Maté programs are small (tens or a hundred bytes), code can still propagate rapidly across large multi-hop networks (tens of seconds).

3.3.1 Propagation and Security

Self-replicating code poses network security risks. Specifically, if an adversary can introduce a single copy of a malicious program, he can take control of the entire network. Maté's version numbers are finite; installing one with the highest possible version number would prevent reprogramming.

A Maté VM can provide two additional levels of security, both of which assume a trusted PC where users write scripts. In the first, motes are physically secure. Private key cryptography can compute packet checksums (authentication codes), with TinySec [12] or similar protocols. The private key is installed as part of the Maté VM.

If some motes can be physically compromised, the PC computes digital signatures using a variant of the BiBA algorithm [22], which provides signatures that are computationally intensive to produce, but inexpensive to verify. Motes maintain one-way hash chains (stored in EEPROM) for validating signatures, which allow them to verify whether the PC generated a given program. This scheme can be attacked by isolating some nodes from the network and observing the hash chains of the rest of the network. However, unisolated nodes will reject the malicious program, as it would use expired hash values. We have incorporated this level of security in Maté as

```
<VM NAME="KNearRegions" DIR="apps/RegionsVM">

<LANGUAGE NAME="tscript">

<FUNCTION NAME="send">
<FUNCTION NAME="mag">
<FUNCTION NAME="cast">
<FUNCTION NAME="id">
<FUNCTION NAME="sleep">
<FUNCTION NAME="KNearCreate">
<FUNCTION NAME="KNearGetVar">
<FUNCTION NAME="KNearPutVar">
<FUNCTION NAME="KNearReduceAdd">
<FUNCTION NAME="KNearReduceMaxID">
<FUNCTION NAME="locx">
<FUNCTION NAME="locy">

<CONTEXT NAME="Boot">
```

Figure 6: Minimal description file for the regions VM shown in Figure 12.

a proof of concept, but it is not part of the distribution: this degree of security is not a pressing requirement in current deployments.

3.4 Building a Maté VM

To build a VM and scripting environment, a user specifies three things: a language, a set of functions, and a set of contexts. From this specification, the Maté toolchain generates a TinyOS component implementing the VM and a Java classes for the assembler. Figure 6 shows the description file for a minimalist abstract regions VM, which we discuss further in Section 4.4.

A Maté supported language supplies the set of primitives it needs to the Maté toolchain. For example, motlle includes primitives to build closures and read local variables, while TinyScript has primitives that read named shared variables. Generating an assembler separates language compilation from the framework. For example, the TinyScript compiler invokes the Maté assembler to produce VM-specific opcodes. Two different VMs that provide the same language may have different instruction to opcode mappings.

3.4.1 Customizing a VM to an application domain

The current Maté framework comes with a collection of 14 contexts (e.g., timers) and 80 functions (e.g., sensor access). These are fairly small, they range in size from 20 to 140 lines of nesC. Each context and function is accompanied by an XML-like specification providing additional information to the Maté toolchain, such as number of arguments to functions and user documentation. A user building a VM can add new contexts and functions by writing additional nesC components implementing the appropriate interfaces (Bytecode from Figure 4 for functions, and an interface to the concurrency manager for contexts).

```
buffer packet;

call bclear(packet);
packet[0] = call light();


call send(packet);
```

(a) TinyScript

```
bpush 3
bclear
light
pushc6 0
bpush 3
bwrite
bpush 3
send
```

(b) Maté Bytecodes

Figure 7: TinyScript function invocation on a simple sense and send loop. *The operand stack passes parameters to functions. In this example, the scripting environment has mapped the variable "packet" to buffer three. The compiled program is nine bytes long.*

4. EVALUATION

We briefly present TinyScript and motlle the two languages Maté currently supports (Section 4.1), measure the basic overheads of interpretation and concurrency (Section 4.2), and verify the effectiveness of code propagation (Section 4.3).

We then evaluate the effectiveness of Maté-based VMs versus the two previous implementation of high-level sensor network programming models, abstract regions (Section 4.4) and TinyDB (Section 4.5).

4.1 Languages

Maté currently supports two languages, TinyScript and motlle. TinyScript is a BASIC-like imperative language with dynamic typing and a simple data buffer abstraction. Everything is statically named: it has neither data nor function pointers. This makes the resource analysis for concurrency straightforward: the resources accessed by a handler are simply the union of all resources accessed by its operations. Static naming allows easy incremental code updates: TinyScript has a one to one mapping between handlers and capsules. TinyScript represents a bare-bones language that provides minimalist data abstractions and control structures that can be compiled to very concise code. Figures 7, 14 and 15 contain TinyScript samples.

The full TinyScript primitive set contains 39 operations, and leaves space in the opcode set for up to one hundred functions. The simplest TinyScript VM (a single context and no functions) uses 1.2kB of RAM and 26kB of code; this includes the TinyOS networking stack, which is used to propagate code.

Motlle (MOTe Language for Little Extensions) is a dynamically-typed, Scheme-inspired language with a C-like syntax. Examples of mottle code are shown in Figures 17, 18 and 19, these are heavily commented to introduce motlle's features by example. The main prac-

	Monolithic	Decomposed	Overhead
Operations/sec	10173	9583	6%

Figure 8: Instruction Issue Rates for Maté. ***The customizability and race condition safety of decomposed VMs imposes a 6% execution overhead.***

Operation	Cycles	Time (μs)
Lock	32	8
Unlock	39	10
Check Runnability	929	232
Run	1077	269
Resume	2038	510
Analysis	15158	3790

Figure 9: Synchronization Overhead. ***Times assume a 4MHz clock.***

tical difference with TinyScript is a much richer data model: motlle supports vectors, lists, strings and first-class functions. This allows significantly more complicated algorithms to be expressed within the VM, but the price is that accurate data analysis is no longer feasible on a mote. To preserve safety, motlle serializes the execution of all event handlers by reporting to the concurrency manager that all contexts access the same shared resource. Thus motlle is not appropriate for applications with bursty processing requirements or which require very rapid response to events. Motlle code is transmitted in a single capsule which contains all handlers; it does not support incremental changes to running programs.

A basic motlle VM, including functions to manipulate lists, strings and vectors takes 29kB of code and 1.9kB of RAM (of which 1kB is available to user programs).

4.2 CPU Overhead

We measured the bytecode interpretation overhead Maté imposes by writing a tight loop and counting how many times it ran in five seconds. Figure 8 shows the results. The loop accessed shared variables (which involve lock checks through the concurrency manager). Maté can issue just under ten thousand instructions per second. This is only 6% slower than the previous published, non-configurable monolithic version.

The nature of Maté operations, especially functions, makes this overhead fairly minor. For example, a function that sends a packet imposes approximately 600 clock cycles of CPU overhead over an operation that consumes 20,000 clock cycles. We believe that VMs will often have even higher level functions, reducing the overhead further. For example, in the RegionsVM described below, creating a region – which transmits several packets and receives many – is a single opcode. Clearly, implementing complex mathematical codes in Maté is inefficient; if an application domain needs significant processing, it should include them as functions.

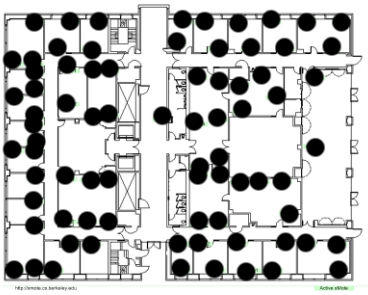

Figure 10: Mote network layout in Soda Hall.

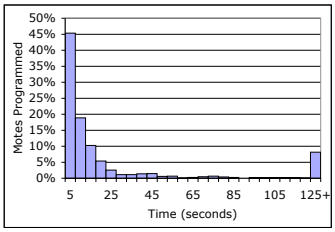

Figure 11: Reprogramming time distribution for a one hundred byte capsule in the Soda Hall network.

We measured the computation overhead of Maté synchronization operations. We measured these using the cycle counter of our mote platforms. Our results are summarized in Figure 9. All values are averaged over fifty samples. These measurements were on a VM with twenty-four shared resources and 128 byte long programs.

Check Runnability is the cost of checking whether the locks a context requires, seeing if each of them is either already held or can be obtained. Running is the cost of obtaining all of a context's unheld locks and posting a TinyOS task to begin execution. Resuming is the cost incurred when a context is triggered to run by an event – this involves checking if the task is runnable (it has or can obtain all of its locks), and if so, running. In this experiment, the context could always run (there was no parallelism), therefore is effectively the sum of Check if runnable and Running with some additional overhead. Installing is the cost of installing a new TinyScript capsule on the mote, which necessitates a full program analysis.

4.3 Propagation

To measure the Maté's code propagation rate, we deployed a simple VM on a 75 mote testbed on the fourth floor of Soda Hall at UC Berkeley. Figure 10 shows the physical topology; the network topology was approximately eight hops across, with four hops being the average node distance. We used the standard Maté pa-

rameters for Trickle. Status and version packets have a τ range of one second to twenty minutes, and a redundancy constant of 2. Fragments use Trickle for suppression, but operate with a fixed window size of one second. The request timeout was five seconds. We injected a one hundred byte (four fragment) program into a single node over a wired link, then measured the time to reception for all of the nodes in the network. We repeated this experiment twenty times, letting the trickles settle between tests. Figure 11 shows the aggregate results.

Forty-five percent of the nodes received the capsule within five seconds, and ninety percent within one minute. The long tail (8% took over two minutes) of the distribution is characteristic of Trickle; because it transmits so few packets to keep the average case inexpensive, a small number of nodes are left behind. However, as the VM continues to trickle the code indefinitely, these nodes will eventually reprogram: after four minutes, on average only four percent (three nodes) still required the program. However, with the parameters described above, a stable node sends at most three packets/hour.

4.4 Regions

We evaluate Maté and abstract regions using a pursuer-evader tracking application. We compare this application implemented in native regions code, in a VM extended with region primitives (RegionsVM) and in a VM customized for pursuer-evader tracking (PegVM).

4.4.1 Pursuer-Evader Tracking

In the Pursuer-Evader Game demo (PEG), a dense field of motes poll their magnetometers several times a second to detect an evader robot. Nodes smooth the readings with an exponentially weighted moving average to filter out transient noise. When a mote detects the evader move by (the filtered reading goes over a threshold), it broadcasts the reading. Because the field is dense, nearby nodes will also detect the evader and broadcast their readings. These broadcasts are an implicit leader election algorithm: the leader is the mote with the highest reading. After broadcasting, a mote waits a short time before deciding whether it is the leader (it did not hear a reading higher than its own). The leader aggregates all of the readings it heard into a single packet and sends it to a waypoint for forwarding to a moving pursuer robot. All in all, the nesC version of the PEG demo took a half dozen students and staff on the order of a month, working full time, to implement and deploy.

4.4.2 RegionsVM

Regions based programming is a recent proposal to simplify sensor network programming [25]. In this model, nodes operate on region-based shared tuple spaces, where regions can be based on geographic proximity, network connectivity, or other node properties. In the proposed programming model, TinyOS provides a single synchronous execution context (a "fiber"), which supports blocking operations. Users compile a full TinyOS image for each regions program, and install the binary image in the network.

Building a RegionsVM (Figure 6) to provide support for regions-based programming is fairly simple: all it requires is writing components that present the regions abstractions as VM functions. Doing so is approximately 400 lines of nesC code per region type. By moving regions into the Maté framework, the user is no longer constrained to a single execution context, as is the case in the proposed regions fiber model: RegionsVM can respond to any execution event for which a Maté context exists. Additionally, instead of retasking by installing a new TinyOS image, the user installs a single image and injects small VM programs.

Figure 12 shows regions pseudocode for the PEG application from the publication by Welsh and Mainland, with corresponding TinyScript code alongside it. The major distinctions stem from the TinyScript data model: it does not support structures or object-oriented style functions. Additionally, the parameter to the creation function is a compile-time constant in the regions compiler. The Maté implementation is seventy-one bytes long.

Figure 13 shows the relative sizes of the two TinyOS images. [2] Maté doubles the size of the executable and adds seven hundred and fifty bytes of data storage. The breakdown of the data storage costs is shown in Figure 13. Buffers and variables are responsible for approximately a third of the storage. The three costs that scale with the number of contexts are Capsule Store (needs space to store programs), Contexts (need to allocate a context), and Variables (each context has a set of private variables).

Maté doubles the size of the TinyOS image, but this becomes a one-time cost for a wide range of regions programs. Instead of sending tens of kilobytes of data into a network to retask it, the user can send on the order of a hundred bytes, a 99% reduction. Of course, the user is also free to install a new VM on the network if a new level of abstraction is needed. Additionally, Maté programs run in the sandboxed VM environment, so buggy code cannot crash the network.

4.4.3 PegVM

The PegVM is a Maté VM with functions for sens-

[2] To compile these applications, we modified many of the standard allocation constants; the default regions settings precluded us from installing it on a mote (it was designed for TOSSIM [15]). Specifically, we set REQ_QUEUE_LEN from 10 to 4, TUPLE_SPACE_MAX_KEY from 32 to 16, ROUTE_TABLE_SIZE from 16 to 8, MHOP_QUEUE_SIZE from 8 to 4, and SEND_QUEUE_SIZE from 32 to 12.

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region_create(8);

while(true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node...  */
    if (max_id == my_id) {
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

(a) Regions Pseudocode

```
!!  Create nearest neighbor region
call KNearCreate();

for i = 1 until 0
  reading = call cast(call mag());

  !!  Store local data as shared variables
  call KNearPutVar(0, reading);
  call KNearPutVar(1, reading * call LocX());
  call KNearPutVar(2, reading * call LocY());

  if (reading > threshold) then
    !!  ID of the node with the max value
    max_id = call KNearReduceMaxID(0);

    !!  If I am the leader node
    if (max_id = my_id) then
      sum = call KNearReduceAdd(0);
      sum_x = call KNearReduceAdd(1);
      sum_y = call KNearReduceAdd(2);
      buffer[0] = sum_x / sum;
      buffer[1] = sum_y / sum;
      call send(buffer);
    end if
  end if
  call sleep(periodic_delay);
next i
```

(b) TinyScript Code

Figure 12: Regions Pseudocode and Corresponding TinyScript. ***The pseudocode is from "Programming Sensor Networks Using Abstract Regions." The TinyScript program on the right is 71 bytes long.***

```
buffer bcastBuf;
shared reading;
private curr;

!!  Read the magnetometer sensor
curr = call cast(call mag());
!!  Filter the reading with an EWMA
reading = (reading * 7) / 8;
reading = reading + curr / 8;
!!  If we have detected a vehicle
if (reading/100) > 200 then
!!  Broadcast our ID and reading
  bcastBuf[0] = call id();
  bcastBuf[1] = reading;
  call bcast(bBuf);
  !!  Fire Timer1 in 500ms
  call settimer1(500);
end if
```

(a) Timer0 (32 bytes)

```
buffer sendBuf;
shared reading;
shared high;

!!  Stop Timer1
call settimer1(0);
!!  Are we are leader?
if reading > high then
!!  If buffer full, use head
  if call bfull(sBuf) then
    sendBuf[0] = call id();
    sendBuf[1] = reading;
!!  Otherwise just append
  else
    sendBuf[] = call id();
    sendBuf[] = reading;
  end if
  call send(sendBuf);
end if
!!  Clear state for next aggregation
call bclear(sendBuf);
high = 0;
```

(b) Timer1 (34 bytes)

```
buffer recvBuf;
buffer sendBuf;
shared high;

!!  Get received data
recvBuf = call received();
!!  Higher than current high?
if (recvBuf[1] > high) then
  high = recvBuf[1];
end if
!!  Add reading to buffer
if (!call bfull(sendBuf) then
  sendBuf[] = recvBuf[0];
  sendBuf[] = recvBuf[1];
end if
```

(c) Receive BCast (52 bytes)

Figure 14: An example PEG implementation in Maté. ***Timer0 fires periodically and samples the magnetometer sensor, using an exponentially weighted moving average to smooth out transient noise. If it detects a spike in the reading, it marks that it sensed something, broadcasts a message to local neighbors, and schedules the aggregation timer (Timer1) to fire in 500ms. When a node hears a broadcast, it puts the heard value into its send buffer and keeps track of the highest reading heard. Timer1 stops itself (it's a one-shot timer) and checks if it has sensed the highest reading in its neighborhood (is the leader). If so, it routes the aggregate buffer to the pursuer with the send() function.***

	Static	Maté
Program (Flash)	19K	39K
Application	1.1K	21K
Regions	5.1K	5.1K
Network Stack	8.5K	8.5K
Timers	1.9K	1.9K
Other	2.5K	2.5K
Data (RAM)	2255	3017
Application	87	741
Regions	1292	1292
Network Stack	572	572
Timers	120	160
Other	194	252

Component	Data
Buffers	176
Capsule Store	211
Variables	96
Contexts	89
Scheduler	56
Locks	48
Other	73
Total	741

Figure 13: Resource utilization of static and Maté TinyOS regions images in bytes.

```
buffer bBuf;
shared reading;
private curr;
curr = call cast(call mag());
avg = (avg * 7) / 8;
avg = avg + curr;
if (avg / 100) > 200 then
bBuf[0] = call id();
bBuf[1] = avg;
call bcast(bBuf);
call settimer1(1);
end if
```

(a) Timer0 (32 bytes)

```
buffer bBuf;
shared reading;

reading = call msense();
bBuf[0] = call id();
bBuf[1] = reading;
call bcast(bBuf);
call settimer1(1);
```

(b) MagSense (14 bytes)

Figure 15: Moving the PEG Event Boundary. ***Timer0 polls the sensor, while MagSense runs when an underlying TinyOS detection implementation fires an event.***

ing, and local and multi-hop communication, designed to support pursuer-evader tracking. Figure 14 shows the corresponding PEG implementation. PegVM could also be more specialized, e.g., averaging and detecting the magnetometer pulses could be an event, as shown in Figure 15. Factoring out the sampling logic makes the code simpler and shorter.

4.4.4 Comparison

To evaluate how accurately the PegVM and the RegionsVM implementations estimate the position of a pursuer, we set up a TOSSIM environment with a configuration similar to the PEG deployment. We arranged nodes in a 10x10 grid, with a spacing of two feet, with packet loss rates drawn from TOSSIM's empirical model [3]. We used a TOSSIM packet-level simulation which models media access and collisions, with a maximum communication rate of 40 packets/second (approximately what current motes are capable of).

In the real PEG deployment, the sensing threshold range was just under one grid spacing, so we modeled the pursuer as a point sensor data source with quadratic strength

[3]The real PEG deployment had a spacing of two yards, not feet. We chose two feet so the overall network density would be identical to the simulation results Welsh and Mainland reported for their regions implementation.

	Packets/event	Estimation error
PegVM	4.5	0.6
RegionsVM (10x bandwidth)	37.5	large
RegionsVM (100x bandwidth)	37.5	0.25

Figure 16: Maté PEG implementation

dropoff over distance, where the threshold was just under two feet. We placed the pursuer at random points within the grid and compared the center of mass estimation from the aggregated readings with the actual position.

Figure 16 shows the results. The PegVM implementation estimated the point source to within six tenths of a foot (a quarter of a grid space), with a average cost of four and a half transmissions per detection event.

Initial attempts to run PEG in RegionsVM failed. A reduction operation requires approximately 40 transmissions. Each node sensing the pursuer performs one reduction, and the leader performs four. With a sampling rate of 2Hz, this is approximately ten times the available network bandwidth. This behavior precluded us from evaluating RegionsVM empirically. Increasing the available bandwidth ten-fold, we observed very large errors (roughly eight feet, sometimes as much as twenty). This was due to the formulation of the program; as each reduction is separate, losses are independent and can skew results. For example, if the request for a large reading is lost in the sum reduction, but not the X or Y coordinate reductions, the divisor will be low when computing the centroid; the formulation of the program ignores this data dependency. A multi-value reduction would remove the possibility of this error, but the current regions library does not support this operation.

We increased bandwidth another ten-fold (to 4,000 packets/second). RegionsVM was able to compute the centroid very accurately, within 0.25 feet. This increased accuracy comes directly from the regions implementation's increased communication. The PegVM implementation, like the actual deployment, only aggregates across values over the threshold. In contrast, the RegionsVM aggregates across all values, but only nodes over the threshold perform the aggregation. Using a push-based instead of a pull-based tuple space implementation would probably significantly reduce the bandwidth required.

4.5 TinyDB

TinyDB [18] is a complex mote application that allows a mote-based sensor network to be treated as a streaming database. This database can be queried using a SQL-like language, TinySQL, whose main extension over SQL is the specification of a 'sample period' at which the query is repeated. For instance, "SELECT nodeid, light SAMPLE PERIOD 60s" will collect the identity and light sensor reading from all nodes in a sensor network every minute. TinySQL supports both simple data collection as

```
/* equivalent TinyDB query:
   SELECT nodeid, parent, light SAMPLE PERIOD 60s */

settimer0(600);       // Fire Timer0 every epoch (60s)
mhop_set_update(120); // Update multihop route every 2min

// We define the Timer0 handler by assigning a function
// to global variable 'timer0_handler'
timer0_handler = fn ()
  // 'mhop_send' sends a message (string) over the multihop
  // network
  // 'encode' encodes the contents of a vector as a string
  // 'next_epoch' advances to the next epoch (except if
  // snoop_epoch made us advance since the last call)
  mhop_send(encode(vector(next_epoch(), id(), parent(),
                          light())));

// The Intercept and Snoop handlers are run when a multihop
// message passes through (intercept_handler) or is overhead
// (snoop_handler) by this mote. If the message is from a
// future epoch, we advance our own epoch.
mhop_snoop_handler = fn () heard(snoop_msg());
mhop_intercept_handler = fn () heard(intercept_msg());
heard = fn (msg)
  {
    // 'decode' decodes a string into the argument vector
    // In this case, the first 2 bytes of the string are
    // decoded into an integer.
    vector v = decode(msg, vector(2));

    // 'snoop_epoch' advances to epoch v[0] if it's beyond ours
    // This makes the network converge on a consistent epoch.
    snoop_epoch(v[0]);
  };
```

Figure 17: "Simple": Simple Data Collection Query in Motlle

```
/* TinyDB:
SELECT nodeid, expavg(temp, 14) WHERE light > 920
       GROUP BY nodeid SAMPLE PERIOD 60s
*/
settimer0(600);       // Fire Timer0 every epoch (60s)
mhop_set_update(120); // Update multihop route every 2min

// 'expdecay(f, b)' builds a function which on every invocation
// evaluates f() and returns its exponentially decaying average
// (with constant 1-2^-b)
expdecay = fn (function attr, int bits)
  {
    int running = 0; // maintains the running total

    // Return a function that samples and decays attr()
    fn ()
      running = running - (running >> bits) + (attr() >> bits)
  };

decaytemp = expdecay(temp, 2);

timer0_handler = fn  ()
  {
    next_epoch();
    if (light() > 100)
      mhopsend(vector(epoch(), nodeid(), decaytemp()));
  };

// decode messages heard and update epoch if necessary
snoop_handler = fn () heard(snoop_msg());
intercept_handler = fn () heard(intercept_msg());
heard = fn (msg)
  snoop_epoch(decode_message(msg, 2)[0]);
decode_message = fn (msg)
  decode(msg, vector(2, 2, 2));
```

Figure 18: "Conditional": Conditional, time-averaged query in Motlle

in the previous example, and aggregate queries such as "SELECT AVG(temperature) SAMPLE PERIOD 60s" to measure the average temperature of the network. The computation of aggregate queries is performed in the network, as data is being collected and routed [19].

We have built a custom VM (QueryVM) based on motlle to perform similar data collection tasks. We designed the extensions in QueryVM with the idea that the motlle code should be responsible for what data to collect when, and for message layouts. The VM extensions are responsible for communication, and abstract common patterns necessary to perform TinyDB-like operations. This led to has extensions in three areas (rather than list the primitives and handlers for these extensions in detail, we comment their use in the examples below):

- Multi-hop communication: we provide primitives and handlers to access the same tree-based multi-hop communication layer used by TinyDB [26].
- Epoch handling: TinyDB query results abstract the notion of time into an "epoch". The first result of a query happens in epoch 1, the second in epoch 2, etc. Epoch numbers are included in query results and help support aggregation. To ensure consistent epoch numbers are maximized across the network by snooping on query results. We add epoch-handling primitives to the VM to avoid replicating epoch-handling logic in every program.
- Aggregation: we add primitives to perform the logic necessary for spatial aggregation. We chose to implement time-based aggregates (such as exponential decay) directly in motlle (see Figure 18).

Figure 17 shows the motlle code for the TinyDB "SELECT nodeid, parent, light SAMPLE PERIOD 60s" query. The core of this code is the single line `mhop_send(...)` which advances the epoch, collects data and sends it over the multi-hop network. The QueryVM application takes 38kB of code and 3.0kB of RAM (of which 1kB is for user programs), while TinyDB uses 59kB of code and 2.9kB of RAM.

We evaluate QueryVM by comparing its performance to TinyDB on three queries: "Simple", a simple data collection query (Figure 17), "Conditional", a conditional, time-averaged query (Figure 18) and "SpatialAvg", a spatially-averaged query (Figure 19). Our main metric is average power consumption, as this controls the lifetime of a sensor network. We also report encoded-query (TinyDB) vs program (QueryVM) size.[4] We ran these queries on a network of 22 mica2 motes spread across the Intel Berkeley lab, with the standard mica sensor board. TinyDB and QueryVM used the same multi-hop routing layer and "low-power listening" radio stack [10]. We measured the power consumption of a non-routing mote physically close to the root of the multi-hop network. Its power thus reflects a mote which overhears most traffic but which

[4]We count each message used in to send a TinyDB query as 31 bytes.

Query	TinyDB (size)	TinyDB (mW)	QueryVM (sizes)	QueryVM (mW)
Simple	93	3.0	91	2.7
Conditional	155	*	155	2.1
SpatialAvg	62	3.0	223	2.3

Table 1: Query size and power consumption in TinyDB and QueryVM

sends relatively few messages.

Table 1 presents the results from these experiments. The results from "Conditional" with TinyDB were incorrect (all nodes reporting a result in a given epoch report the an identical value), so we do not include that result. The QueryVM queries were 11% to 30% more energy efficient. For the simple data collection, we estimate that this difference (0.3mW) is due to the larger data packets used by TinyDB. Spatial averaging has at most the same communication cost as simple data collection, so at least 0.4mW of the energy difference must be due to higher computational cost in TinyDB.

It is clear from these examples that a TinyDB query is more concise and easier to write than the corresponding QueryVM code. However, QueryVM is more expressive: it is possible (Figure 18) to measure an exponentially decaying average within QueryVM, something that is only possible with TinyDB if it was provided when TinyDB itself was compiled. Similarly, we debugged our spatial averaging by writing the code in motlle, at the expense of a larger program (533 vs 223 bytes) and 10% more power consumption (2.5 vs 2.3mW). Ultimately, the choice of the appropriate programming abstraction must depend on the needs of the application. Our results show that the functionality of TinyDB can be provided within Maté with comparable cost.

At a power consumption of 2.7mW, a pair of AA batteries (2700mAh, of which approximately 2/3rds is usable by a mote) would last more nearly three months. By lowering the sample rate and other optimizations, we believe that lifetimes of three months or more are readily achievable. This shows that Maté is a realistic option for long term, low-duty-cycle sensor net deployments.

5. DISCUSSION

In the Maté model, user programs have three distinct representations, shown in Figure 1. First is the user representation, where programs are high-level scripts. Maté meets the needs of this level, programming ease and abstraction, by supporting multiple languages. A compiler transforms the user representation into the bytecodes of the transmission representation. These bytecodes are designed for conciseness and safety, allowing networks to freely propagate and install them. Although currently Maté merely interprets the in-network representation, there is no reason it could not use just-in-time compilation techniques to generate native code.

```
/* TinyDB: SELECT avg(temp) SAMPLE PERIOD 60s */
settimer0(600);       // Fire Timer0 every epoch (60s)
mhop_set_update(120); // Update multihop route every 2min

// 'spatialavg(f)' returns an "object" with methods for
// performing spatial averaging of f. The methods are:
//   spatial_sample: measure f locally
//   spatial_agg: include results from a child
//   spatial_get: get completed results for this subtree
//     (as a string of length spatialavg_length, ready for
//     transmission)
// An "object" is a vector with functions as elements.
spatialavg = fn (function attr) {
    any sstate = spatialavg_make(attr);
    epoch_change = fn() spatialavg_epoch_update(sstate);
    vector(fn (data) spatialavg_intercept(sstate, data),
   fn () spatialavg_sample(sstate, sstate[0]()),
   fn () spatialavg_get(sstate))
};

avgtemp = spatialavg(temp);

timer0_handler = fn  ()  {
    any summary;
    // the root (id 0) does only aggregation
    if (id()) {
        next_epoch();
        avgtemp[spatial_sample]();
    };
    summary = avgtemp[spatial_get]();
    if (summary)
      mhopsend(encode(vector(epoch(), summary)));
};

// decode messages heard, update epoch if necessary,
// add child summaries to our summary
snoop_handler = fn ()
  snoop_epoch(decode_message(snoop_msg())[0]);

intercept_handler = fn () {
    vector fields = decode_message(intercept_msg());
    snoop_epoch(fields[0]);
    avgtemp[spatial_agg](fields[1]);
};

decode_message = fn (msg)
  decode(msg, vector(2, make_string(spatialavg_length)));
```

Figure 19: "SpatialAvg": Spatially averaged Query in Motlle

In Section 4, we showed how the Maté architecture can be used to generate VMs that provide proposed regions-based or query-based programming models; these VMs are more efficient than the monolithic implementations they emulate. This improved efficiency comes from how Maté decomposes in-situ programming into three layers. Certainly, an optimized, fully integrated implementation of these models could be made more efficient than one implemented in Maté; however, our results suggest that such implementations would benefit from following a similar layering. Additionally, by decomposing an environment into a set of reusable and robust building blocks, the architecture greatly simplifies the process of developing programming models for new applications.

Sensor networks, TinyOS networks in particular, are notoriously difficult to program. At the level of an individual mote, software is a mix of embedded system and kernel code, with all of the resulting complexities. Current mote hardware does not support traditional protection boundaries, and it is an open question whether future platforms will, for a variety of design considerations, such as stack memory usage. Maté provides the equiv-

alent of a user-land programming environment through an intermediate representation. In this light, building a VM is similar to building a customized OS kernel: primitives and functions are the system calls. Additionally, by tailoring the representation to a particular application domain, Maté can encode programs more concisely than native code, making propagation more efficient and conserving RAM.

6. RELATED WORK

Extensible abstraction boundaries have a long history in operating systems. Proposals such as scheduler activations [2], ACPM [9] and U-Net [24] show that having richer boundaries that allow application-OS cooperation can greatly improve application performance. Operating systems such as exokernel [11] and SPIN [3] take a more aggressive approach, using software structure or language safety to allow flexible OS service composition. Instead of providing a fixed interface, they allow users to write the interface, and improve performance through increased control.

Maté uses similar techniques to make adding extensions easy, but execution performance is not the driving goal behind the architecture's extensibility. Enabling users to easily compose VMs tailored to a specific application results in simple programs and very concise code; this conciseness minimizes overhead. Similar to OSKit [7], defining system boundaries makes these compositions simple and easy. Additionally, making the system building blocks self-contained components is good software engineering practice, as it can localize faults and make bugs much easier to track down; microkernel efforts such as Mach [23] had similar benefits, although, again, for different goals.

Virtual machines such as the UCSD p-System [21], the Java Virtual Machine [17], and Microsoft's CLR [1] provide common abstractions across a wide range of platforms, supporting one or more languages. This contrasts strongly with Maté's goal of providing a virtual machine for a particular deployment. These systems share the advantage of smaller-than-native code with Maté, though we believe that Maté VMs can take advantage of their application-specific nature to provide even more compact code (as the examples from Section 4 where we specialize an existing VM show).

`Vmgen` [5] is another tool for building interpreters, but with a rather different focus than Maté. Its main goals are to simplify the specification of an interpreter's instructions (e.g., addition, pushing constants on a stack, etc) and increase interpreter execution efficiency. In contrast, Maté aims to simplify extending core functionality provided by some language with application-specific extensions, and to provide core system services such as concurrency management and code propagation. The techniques proposed by `Vmgen` could be used to improve the performance of Maté VMs. A number of techniques [13, 6, 4] have been proposed to reduce code size for interpreters and regular processors. These optimizations could be applied to further reduce the size of Maté programs.

7. CONCLUSION

We argue that application specific virtual machines are an effective solution to safe and efficient mote network programming. A VM takes the role of a traditional OS kernel. Instead of system calls, it provides a set of primitives. The VM schedules concurrent VM-level threads, manages shared resources, and enforces protection boundaries. As VMs are application specific, they provide a set of primitives particular to what a user needs, making programs short and simple: high level application logic can be encoded in a few hundred bytes. Because bytecodes sit above high-level operations, the interpretation overhead is small. Additionally, virtualization gives all of its traditional benefits, such as handling platform heterogeneity.

At the same time, several research challenges remain. The framework could provide better mechanisms for type extensions, and benefit from a resource analysis framework capable of supporting languages with more dynamic memory allocation. Power efficiency is an important part of sensor network applications; although Maté addresses the cost of propagation and execution, it should also manage hardware resources such as sensors; a combination of static analysis (like for concurrency) and dynamic resource tracking could automatically enable and disable hardware devices as needed.

Placing clear divisions between the three program layers allows each to be individually optimized for its separate goals. A variety of proposals exist for high level programming models, but sensor network operating systems are highly optimized embedded systems. By providing a architecture to bridge the gap between them, Maté allows users to efficiently and safely reprogram sensor networks.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599 and NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

8. REFERENCES

[1] TC39/TG2. Common Language Infrastructure (CLI). Technical Report ECMA-334, 2001.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, 1995.

[4] J. Ernst, W. S. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 358–365, 1997.

[5] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen — A Generator of Efficient Virtual Machine Interpreters. *Software Practice and Experience*, 32(3):265–294, 2002.

[6] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[9] K. Harty and D. Cheriton. Application controlled physical memory using external page cache management, October 1992.

[10] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.

[11] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[12] C. Karlof, N. Sastry, and D. Wagner. TinySec: Security for TinyOS, 2002. Presentation given at NEST group meeting, 11/21/2002.

[13] M. Latendresse and M. Feeley. Generation of fast interpreters for huffman compressed bytecode. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators*, 2003.

[14] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[15] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[16] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 491–502. ACM Press, 2003.

[19] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[20] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[21] S. Pemberton and M. C. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[22] A. Perrig. The biba one-time signature and broadcast authentication protocol. In *ACM Conference on Computer and Communications Security*, pages 28–37, 2001.

[23] R. Rashid, R. Baron, A. Forin, M. J. David Golub, D. Julin, D. Orr, and R. Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual ACM Symposium on Operating Systems Principles*, December 1995.

[25] M. Welsh and G. Mainland. Programming sensor networks with abstract regions. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[26] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

Appendix: Deadlock-free Proof

The follow proof shows that Maté's concurrency model is deadlock-free.

Proof by contradiction: assume deadlock exists. Represent the invocations as a traditional lock dependency graph, where each context is vertex. For all vertices C_a in the suspended or waiting state, we record the time t_a at which C_a suspended (for newly created invocations, t_a is the invocation creation time). There is a directed edge from C_a to C_b if C_a is waiting on a resource that C_b holds.

An edge represents one of two situations. First, C_a may be a newly created invocation waiting to begin execution. Second, C_b may be holding resources that C_a released after suspending at time t_a but wants to reacquire. Because resources sets are atomically acquired at the end of a scheduling point, it follows that C_b resumed execution after C_a was suspended. Therefore, if C_b suspends at time t_b, we can conclude that $t_b > t_a$. Contexts acquire resource sets atomically, so a context which is waiting to start execution does not hold any resources.

For deadlock, there must be a cycle $C_1, \ldots, C_n$ in the graph. Every context in the cycle must be in the waiting state. A context C_k from the cycle cannot be at its start vertex as it has an incoming edge, therefore the invocation C_j after C_k in the cycle must have $t_k > t_j$. By induction, it follows that $t_1 > t_1$, a contradiction.